COMS W4111: Introduction to Databases
Spring 2024, Sections 002/V02

Midterm

Introduction

This notebook contains the midterm. **Both Programming and Nonprogramming tracks should complete this.** To ensure everything runs as expected, work on this notebook in Jupyter.

- You may post **privately** on Edstem or attend OH for clarification
  - TAs will not be providing hints

Submission instructions:

- You will submit **PDF and ZIP files** for this assignment. Gradescope will have two separate assignments for these.
- For the PDF:
  - The most reliable way to save as PDF is to go to your browser's menu bar and click `File -> Print`. Switch the orientation to landscape mode, and hit save.
  - **MAKE SURE ALL YOUR WORK (CODE AND SCREENSHOTS) IS VISIBLE ON THE PDF. YOU WILL NOT GET CREDIT IF ANYTHING IS CUT OFF.** Reach out for troubleshooting.
- For the ZIP:
  - Zip a folder containing this notebook and any screenshots.
- Further submission instructions may be posted on Edstem.

Setup

In [19]:
```
%load_ext sql
%sql mysql+pymysql://root:dbuserdbuser@localhost
```

In [20]:
```
import pandas
from sqlalchemy import create_engine
engine = create_engine("mysql+pymysql://root:dbuserdbuser@localhost")
```

In [1]:
```
from IPython.display import Image
```

Written

- You may use lecture notes, slides, and the textbook
- You may use external resources, but you must cite your sources
- As usual, keep things short

W1

Briefly explain structured data, semi-structured data, and unstructured data. Give an example of each type of data.

Note: I reviewed the textbook in preparing the answer to this problem.

1. **Structured data** is data that adheres to a strict form/structure and is highly organized. An example is SQL relational databases in which all data is contained in tables.

2. **Semi-Structured data** is data that has some properties of structured data but is more flexible. Typical features of semi-structured data include flexible schema and multivalued data types (such as sets, mappings, and arrays). Examples of semi-structured data include JSON and XML.

3. **Unstructured data** is data that has no fixed/assumed structure or format. Typically information retrieval methods such

as keyword search are used to query unstructured data. Examples include textual data such as emails, word documents and webpages on the web.

W2

Codd's 0th rule states:

```
For any system that is advertised as, or claimed to be, a relational database management
system, that system must be able to manage databases entirely through its relational
capabilities.
```

Briefly explain and give examples of how the rule applied to:

1. Metadata
2. Security

Codd's 0th rule means that the relational database must be able to enforce/implement whatever it requires by using tables and its relational capabilites, and without appealing to outside software applications. All information, including that for metadata and security, must therefore be stored in a table.

1. **Metadata**: Per Codd's 0th rule, the relational database management system stores **metadata (data about the data/ schema stored in the database) in the form of tables**. The metadata information can be accessed, changed and maintained through the relational capabilities of the DBMS. Metadata includes table names, attribute types, integrity constraint (and in general all definition language) all of which the database must store in a table.

2. **Security**: Per Codd's 0th rule, the relational database management system stores the **secutity information in tables** and uses them to enforce security on its own by its relational capabilities. The secutity information (users, privileges of different kinds etc..) are therefore stored in tables (e.g. under information_schema).

   In relational DBMS such as SQL, these tables are used to achieve its security goals. For example the tables can be used to grant any or none of select, insert, update, delete privileges and/or ability to modify the schema (Authorization). Further, SQL the tables to perform authentication. Additionally, SQL allows creation of views (using tables) to further control which portion of the data a user has access to.

W3

Codd's 6th rule states:

```
All views that are theoretically updatable are also updatable by the system.
```

Using the following table definition, use SQL (`create view`) to define

```
1. Two views of the table that are not possible to update
2. One view that is possible to update
```

You do not need to execute the statements. We are focusing on your understanding.

```
create table student
(
    social_security_no char(9) not null primary key,
    last_name varchar(64) null,
    first_name varchar(64) null,
    enrollment_year year null,
    total_credits int null
);
```

Note: I reviewed the lecture slides and textbook in answering this question. A view can only be updated when it can be translated back to the underlying schema unambiguously/uniquely. More formally (from the lecture notes/book), a view can only be updated when:

- The from clause has only one database relation.
- The select clause contains only attribute names of the relation, and does not have any expressions, aggregates, or distinct specification.
- Any attribute not listed in the select clause can be set to null
- The query does not have a group by or having clause.

1. The following are two views that are **not possible** to update:

```
create view impossible_1 as
select last_name,first_name,total_credits from student;
```

The view `impossible_1` cannot be updated as the attribute `social_security_no` is not listed in the select clause of the view and cannot be set to null.

```
create view impossible_2 as
select * from student join takes using (ID);
```

Letting `takes` be a table of courses taken by student (as in the db_book schema), the join operation in the `from` clause prevents the view `impossible_2` from being updatable. The view cannot be updated because the from clause has more than one database relation.

3. The following is a view that **is possible** to update:

```
create view possible as
select social_security_no,last_name,first_name from student;
```

The view `possible` can be updated as any update can be translated back to the underlying table student unambiguously.

More formally, the query is updatable as: the from clause only has one table, the select clause only has attribute names of the table without specification or aggregation, the attributes `enrollment_year` and `total_credits` not in the select clause can be set to null, and the query does not have any groupby operations.

W4

The Columbia University directory of courses uses `20241COMS4111W002` for this sections "key".

1. Is this key atomic? Explain.
2. Explain why having non-atomic keys creates problems for indexes.

1. The key is not atomic. A key is atomic when the value has no subparts/is indivisible. In our example (as in lecture) the key `20241COMS4111W002` can be divided into several subparts:

- year: '2021'
- semester code: '1' (for Spring)
- department code: 'COMS'
- course code/number: '4111'
- faculty code: 'W'
- section code/number: '002'

Note: I refered to https://backend.turing.edu/module2/lessons/many_to_many for the "difficulty in checking" part of (2).

2. A non-atomic key (e.g. a key on a column with non-atomic value) creates problems for indices as they are harder to check/maintain and make it harder to implement foreign key constraints for a subpart.

   - **Difficultity in Maintaining**: Whenever a portion of the key changes, any item referencing the key must be updated. The entire key is updated wherever referenced even if that particular usage only needs a portion of the key. For example, if a course requires COMS4111 as a prereq and uses a foreign key on `20241COMS4111W002`, it would be problematic as every semester the key would need to be updated.

   - **Difficulty in Checking**: Checking the constraint will be harder as all of the subparts must be checked against each other at once. It would be more efficient to check a subpart individually. As the type of the non-atomic column will be string (and longer than a subpart), the storage will be more costly than if the database knew the types of the subparts (which may be smaller than the max string length). The increased storage size makes checking/enforcing constraints more difficult.

   - **Difficulty being referenced by Foreign keys**: Another problem with a non-atomic column is that foreign keys referring to a subpart will run into difficulties and require data engineering to achieve their result. For example, a foreign key referring to course codes would need to extract the relvent subpart (e.g. `4111`).

W5

Briefly explain the following concepts:

1. Natural join
2. Equi-join
3. Theta join
4. Left join
5. Right join
6. Outer join
7. Inner join

A JOIN operation combines the tables "side by side" by combining the rows together accross columns from both tables. Depending on the type of JOIN, only matching rows are used (inner JOIN), all rows from one table and matching rows from the other (LEFT/RIGHT OUTER JOIN), or all the rows of both tables are used (FULL OUTER JOIN). The method of matching is determined by the type of join condition (Natural, Equi or Theta). Which columns are used is determined by the type of JOIN used and the join condition.

1. **Natural join** combines rows that match on all shared columns/attributes. A single copy of the shared columns is retained. In SQL the "using" keyword allows specification of which columns to use as the shared attribute. A natural join can be applied to both inner and outer forms of join. In SQL, an unqualified Natural join is a Natural Inner Join.

2. **Equi-join** performs a join that combines two tables when specified attributes from the tables are equal. For example, the equi-join condition A.a=B.b gives the join of tables *A* and *B* where attribute *a* from table *A* equals the value of attribute *b* from table *B* (*a* and *b* may also be sets of attributes). Unlke a Natural join, the attributes *a* and *b* need not have the same name. Further, unlike a Natural join both column *a* and *b* are retained in the output. An equi-join can be applied to both inner and outer forms of join.

3. **Theta join** is an extension of Equi-join to join two tables A and B using a condition θ to include all rows satisfying θ (and discarding all other rows) where θ need not be "=". Formally a Theta join is defined as $A \bowtie_\theta B = \sigma_\theta(A \times B)$ where θ is any predicate on attributes in $A \cup B$. For example, θ can be of the form A.budget>B.price to give a join of tables A and B where the budget attribute from table A is greater than the price attribute from table B. Theta join can be applied to both inner and outer forms of join.

4. **Left join** is a form of outer join that preserves all rows of the Left table using either no condition or any of Natural, Equi-join or Theta join as the join condition for matching. The unmatched Left table rows are populated with Null values for the Right table's attributes.

5. **Right join**, parallel to Left join, preserves all rows of the Right table using either no condition, or any of Natural, Equi-join or Theta join as the join condition for matching. The unmatched Right table rows are populated with Null values for the Left table's attributes.

6. **Outer join** is a form of join between two tables that (depending on which form) preserves unmatched rows in one or both of the tables. In an outer join, the output has the attributes from both tables. The unmatched preserved rows are populated with Null values for the other table's attributes. An outer join is either Left join (preserving all rows of the Left table), Right join (preserving all rows of the Right table) or Full Outer join (preserving all rows of both tables). An outer join can use any of Natural, Equi-join or Theta join as the join condition for matching.

7. **Inner join** is a join operation between two tables that only preserves the matching rows (in contrast to outer join) and discards all unmatched rows. An inner join can use any of Natural, Equi-join or Theta join as the join condition for matching.

W6

The *Classic Models* database has several foreign key constraints. For instance, *orderdetails.orderNumber* references *orders.orderNumber*.

1. Briefly explain the concept of *cascading actions* relative to foreign keys.
2. How could cascading actions be helpful for the above foreign key relationship?

Note: I reviewed the textbook in answering this question.

1. Cascading action determine what should happen to foreign keys when the item being referenced is altered. Cascading actions include delete and update. For example, deleting an order number would "cascade" to delete any row in orderdetails with a foreign key referencing the deteled order number. Similarly, updating an order number would "cascade" to update the value of all rows of orderdetails with a foreign key referencing the updated order number.

2. For the above foreign key relationship, cascading actions could be helpful in the event that an order is cancelled and deleted from the database: the cascading action would remove all the rows of orderdetails with a foreign key referencing orderNumber.

An additional case in which cascading would be helpful is when an orderNumber is changed: the associated orderdetails rows would be updated appropriately. A use case would be when there was a clerical error in the order and a new order number is issued with the correction.

W7

Give two reasons for using an associative entity to implement a relationship instead of a foreign key.

Note: For part of (1) I reffered to https://backend.turing.edu/module2/lessons/many_to_many

1. An associative entity allows implementation of a many to many relationship. A foreign key is insufficient to implement a many to many relationship as it references a single entity in the "foreign" table but not multiple entities.
2. An associative entity allows storage of attributes/properties on the relationship (as opposed to the entities). As in lecture, an example would be having the attribute "data supervision started" on the advisor relationship connecting students and instructors. A foreign key cannot contain properties of the relationship as it can only enforce existence of an attribute value in the another table.

W8

Briefly explain how SQL is closed under its operations. Give a simple query that takes advantage of this.

Every operation in SQL produces a table and therefore SQL is closed under its operations. Any place calling for a table can be replaced by a SQL operation.

A simple query that takes advantage of this is using select on the result of a join operation. For example, using the db_book schema we have the following simple query to get a list of each student and the courses they have taken:

```
select name, course_id
from student
        join takes using (ID)
```

W9

Briefly explain the differences between:

1. Database stored procedures
2. Database functions
3. Database triggers

As in lecture, stored procedures, functions, and triggers differ in (1) their ability to change data, (2) whether they have a return value, and (3) the method in which they are "called".

1. Stored procedures and triggers can change the data in the database, while functions cannot.
2. Functions always return a value, stored procedures may sometimes return a value, and triggers never do.
3. Stored procedures are called by being executed, functions are called in statements, and triggers are not explictly called but rather are "called" as the reaction to an event occuring.

In summary, while database stored procedures, database functions, and database triggers all allow the database to perform imperative "business logic" they differ in the following ways:

	Functions	**Stored Procedure**	**Trigger**
Change data?	❌	✔	✔
Return value?	✔	Sometimes	❌
Method of calling	In statement	Execution	Reaction

W10

List three benefits/use cases for defining views.

1. **Security/Privacy.** As in lecture, we may want to hide salaries from regular users but allow access to other parts of the data.
2. **User applications.** As in lecture, a user or application may expect the data be in a certain format (different than the database relational schema). Using a view allows the data to be presented to the user/application as expected without modifying the underlying database.

3. **Specialization/Inheritance.** As in lecture, a view can be useful in cases of specialization/inheritance for the 1,2, and 3 table methods. Depending on the implementation, a view is used to present the information for the "parent"/"child" tables without needing to store repetitive/redundant information.

Relational Algebra

- Use the Relax calculator for these questions.
- For each question, you need to show your algebra statement and a screenshot of your tree and output.
  - **For your screenshot, make sure the entire tree and output are shown.** You may need to zoom out.
- The suggestions on which relations to use are hints, not requirements.

R1

- Write a relational algebra statement that produces a relation showing **teachers that taught sections in buildings that didn't match their department's building**.
  - A section is identified by `(course_id, sec_id, semester, year)`.
- Your output should have the following columns (names should match exactly; there should be no prefixes):
  - `instructor_name`
  - `instructor_dept`
  - `course_id`
  - `sec_id`
  - `semester`
  - `year`
  - `course_building`
  - `dept_building`
- You should use the `teaches`, `section`, `instructor`, and `department` relations.
- As an example, one row you should get is

instructor_name	instructor_dept	course_id	sec_id	semester	year	course_building	dept_building
'Srinivasan'	'Comp. Sci.'	'CS-101'	1	'Fall'	2009	'Packard'	'Taylor'

- Srinivasan taught CS-101, section 1 in Fall of 2009 in the Packard building. However, Srinivasan is in the CS department, whose building is Taylor.

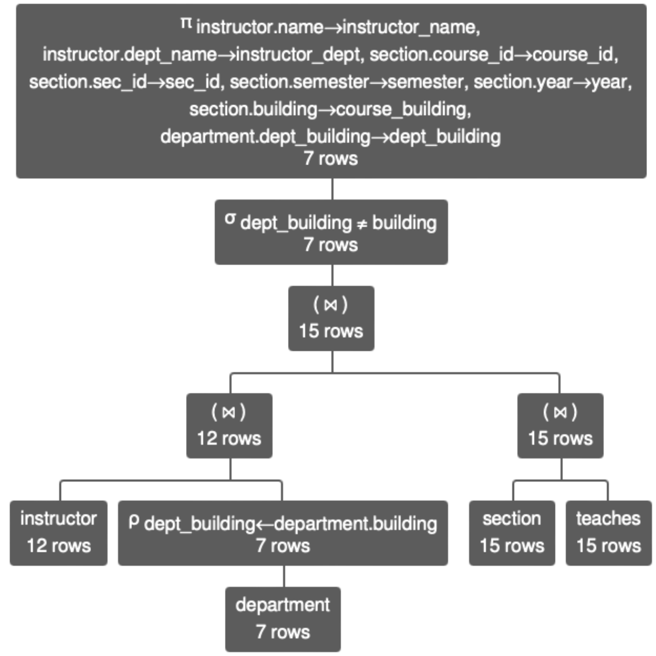
Algebra statement:

```
π
    instructor_name←instructor.name,
    instructor_dept←instructor.dept_name,
    course_id←section.course_id,
    sec_id←section.sec_id,
    semester←section.semester,
    year←section.year,
    course_building←section.building,
    dept_building←department.dept_building
(
σ dept_building≠building (
(instructor⨝(ρ dept_building←department.building department))⨝(section⨝teaches))
)
```

Execution:

```
In [2]: Image("R1.png")
```

Out[2]:



$\pi_{\text{instructor.name}\rightarrow\text{instructor\_name, instructor.dept\_name}\rightarrow\text{instructor\_dept, section.course\_id}\rightarrow\text{course\_id, section.sec\_id}\rightarrow\text{sec\_id, section.semester}\rightarrow\text{semester, section.year}\rightarrow\text{year, section.building}\rightarrow\text{course\_building, department.dept\_building}\rightarrow\text{dept\_building}} (\sigma_{\text{dept\_building} \neq \text{building}} ((\text{instructor} \bowtie (\rho_{\text{dept\_building}\leftarrow\text{department.building}} \text{department})) \bowtie (\text{section} \bowtie \text{teaches})))$

Execution time: 5 ms

instructor_name	instructor_dept	course_id	sec_id	semester	year	course_building	dept_building
'Srinivasan'	'Comp. Sci.'	'CS-101'	1	'Fall'	2009	'Packard'	'Taylor'
'Srinivasan'	'Comp. Sci.'	'CS-315'	1	'Spring'	2010	'Watson'	'Taylor'
'Wu'	'Finance'	'FIN-201'	1	'Spring'	2010	'Packard'	'Painter'
'Katz'	'Comp. Sci.'	'CS-101'	1	'Spring'	2010	'Packard'	'Taylor'
'Katz'	'Comp. Sci.'	'CS-319'	1	'Spring'	2010	'Watson'	'Taylor'
'Crick'	'Biology'	'BIO-101'	1	'Summer'	2009	'Painter'	'Watson'
'Crick'	'Biology'	'BIO-301'	1	'Summer'	2010	'Painter'	'Watson'

|| | :---: | | **R1 Execution Result**|

R2

- Some students don't have instructor advisors. Some instructors don't have student advisees.
- Write a relational algebra statement that produces a relation showing **all valid pairing between unadvised students and instructors with no advisees**.
  - A pairing is valid only if the student's department and instructor's department match.
- Your output should have the following columns (names should match exactly; there should be no prefixes):
  - `instructor_name`
  - `student_name`
  - `dept_name`
- You should use the `advisor` , `student` , and `instructor` relations.
- **You may only use the following operators:** π, σ, =, ≠, ∧ (and), ∨ (or), ρ, ←, ⋈, ⟖, ⟕, ⟗
  - You may not need to use all of them.
  - Notably, you may **not** use anti-join or set difference.
- As an example, one row you should get is

instructor_name	student_name	dept_name
'El Said'	'Brandt'	'History'

- El Said has no advisees, and Brandt has no advisor. They are both in the history department.
- The same instructor may show up multiple times, but the student should be different each time. Similarly, the same student may show up multiple times, but the instructor should be different each time.

Algebra statement:

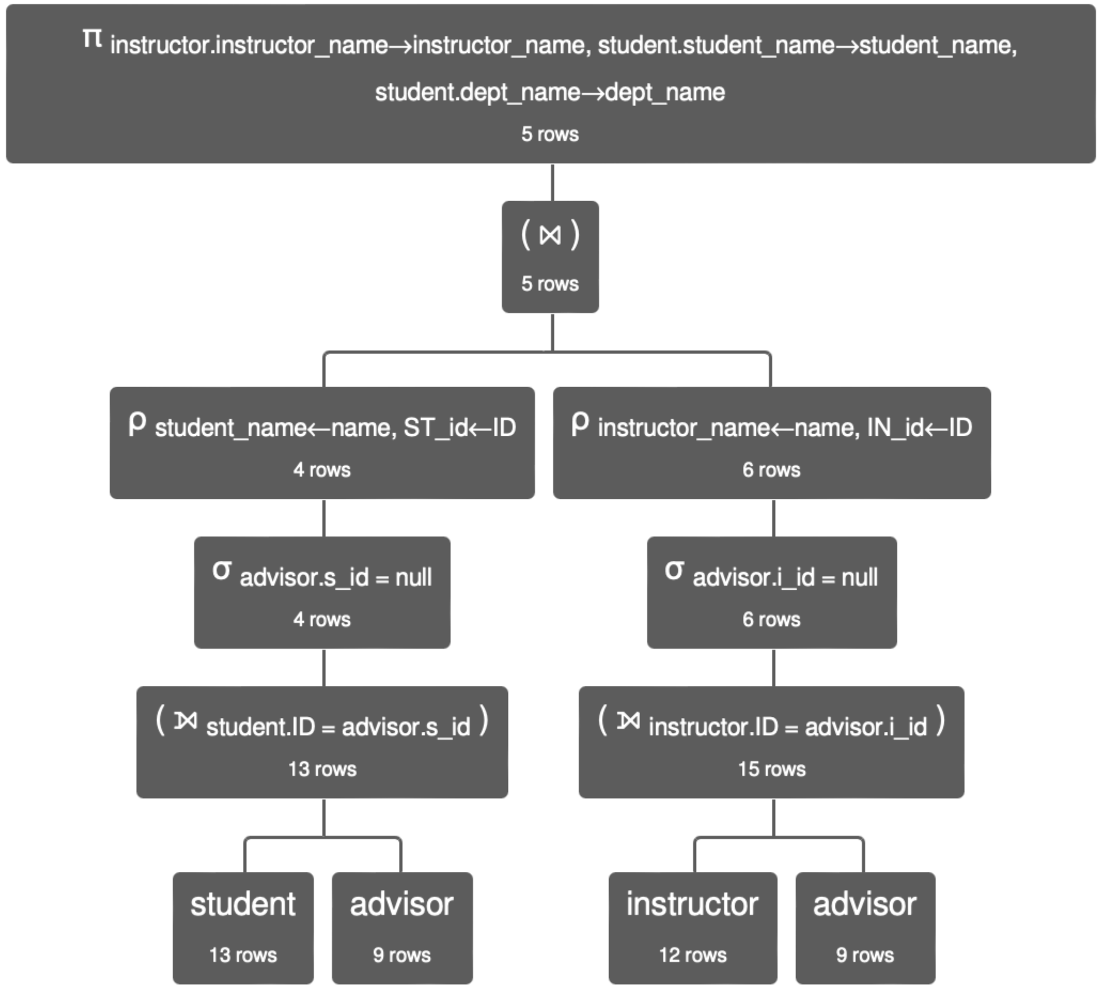
```
π
  instructor_name←instructor.instructor_name,
```

```
  student_name←student.student_name,
  dept_name←student.dept_name
(
ρstudent_name←name,ST_id←ID(σ advisor.s_id=null
(student⟕ student.ID=advisor.s_id advisor))
⋈
ρinstructor_name←name,IN_id←ID(σ advisor.i_id=null
(instructor⟕ instructor.ID=advisor.i_id advisor))
)
```

Execution:

In [5]:
```
Image("R2.png",width=950)
```

Out[5]:



$\pi_{\text{instructor.instructor\_name}\to\text{instructor\_name, student.student\_name}\to\text{student\_name, student.dept\_name}\to\text{dept\_name}} (\rho_{\text{student\_name}\leftarrow\text{name, ST\_id}\leftarrow\text{ID}} (\sigma_{\text{advisor.s\_id = null}} (\text{student} ⟖_{\text{student.ID = advisor.s\_id}} \text{advisor})) \bowtie \rho_{\text{instructor\_name}\leftarrow\text{name, IN\_id}\leftarrow\text{ID}} (\sigma_{\text{advisor.i\_id = null}} (\text{instructor} ⟖_{\text{instructor.ID = advisor.i\_id}} \text{advisor})))$

Execution time: 4 ms

instructor_name	student_name	dept_name
'El Said'	'Brandt'	'History'
'Califieri'	'Brandt'	'History'
'Brandt'	'Williams'	'Comp. Sci.'
'Mozart'	'Sanchez'	'Music'
'Gold'	'Snow'	'Physics'

R2 Execution Result

R3

- Consider `new_section` , defined as:

  `new_section = π course_id, sec_id, building, room_number, time_slot_id (section)`

- `new_section` contains sections, their time assignments, and room assignments independent of year and semester.
  - For this question, you can assume all the sections listed in `new_section` occur in the same year and semester.
  - You should copy the given definition of `new_section` to the top of your Relax calculator and treat it as a new relation.
- Write a relational algebra statement that produces a relation showing **conflicting sections**.
  - Two sections conflict if they have the same `(building, room_number, time_slot_id)` .
- Your output should have the following columns (names should match exactly; there should be no prefixes):

- `first_course_id`
- `first_sec_id`
- `second_course_id`
- `second_sec_id`
- `building`
- `room_number`
- `time_slot_id`

- You should use the `new_section` relation.
- Your output cannot include courses and sections that conflict with themselves, or have two rows that show the same conflict.
- Good news: I'm going to give you the correct output!

first_course_id	first_sec_id	second_course_id	second_sec_id	building	room_number	time_slot_id
'CS-190'	2	'CS-347'	1	'Taylor'	3128	'A'
'CS-319'	2	'EE-181'	1	'Taylor'	3128	'C'

- Bad news: Your output must match mine **exactly**. The order of `first_course_id` and `second_course_id` cannot be switched.
  - Hint: You can do string comparisons in Relax using the inequality operators.

Algebra statement:

```
new_section = π course_id, sec_id, building, room_number, time_slot_id (section)

/* Define new tables to use for mapping course_id to sec_id */
first_section_pairing=π course_id→first_course_id, sec_id→first_sec_id, building,
room_number, time_slot_id (new_section)

second_section_pairing=π course_id→second_course_id, sec_id→second_sec_id, building,
room_number, time_slot_id (new_section)

/* Use groupby building,room_number, time slot id to find all courses that conflict, as
we want alphabetical sorting we use Min and Max on the string course_id to find
```

```
first_course_id and second_course_id. */

ONE=γ building, room_number, time_slot_id; count(course_id)->
num_conflict,Min(course_id)->first_course_id, Max(course_id)->second_course_id
new_section

/* Select courses that conflict */
TWO=σ num_conflict>1 ONE

/* Use join and mappings to get the appropriate sec information */
THREE=TWO⨝ first_section_pairing
FOUR=THREE⨝ second_section_pairing

π
    first_course_id,
    first_sec_id,
    second_course_id,
    second_sec_id,
    building←section.building,
    room_number←section.room_number,
    time_slot_id←section.time_slot_id
                (FOUR)
```

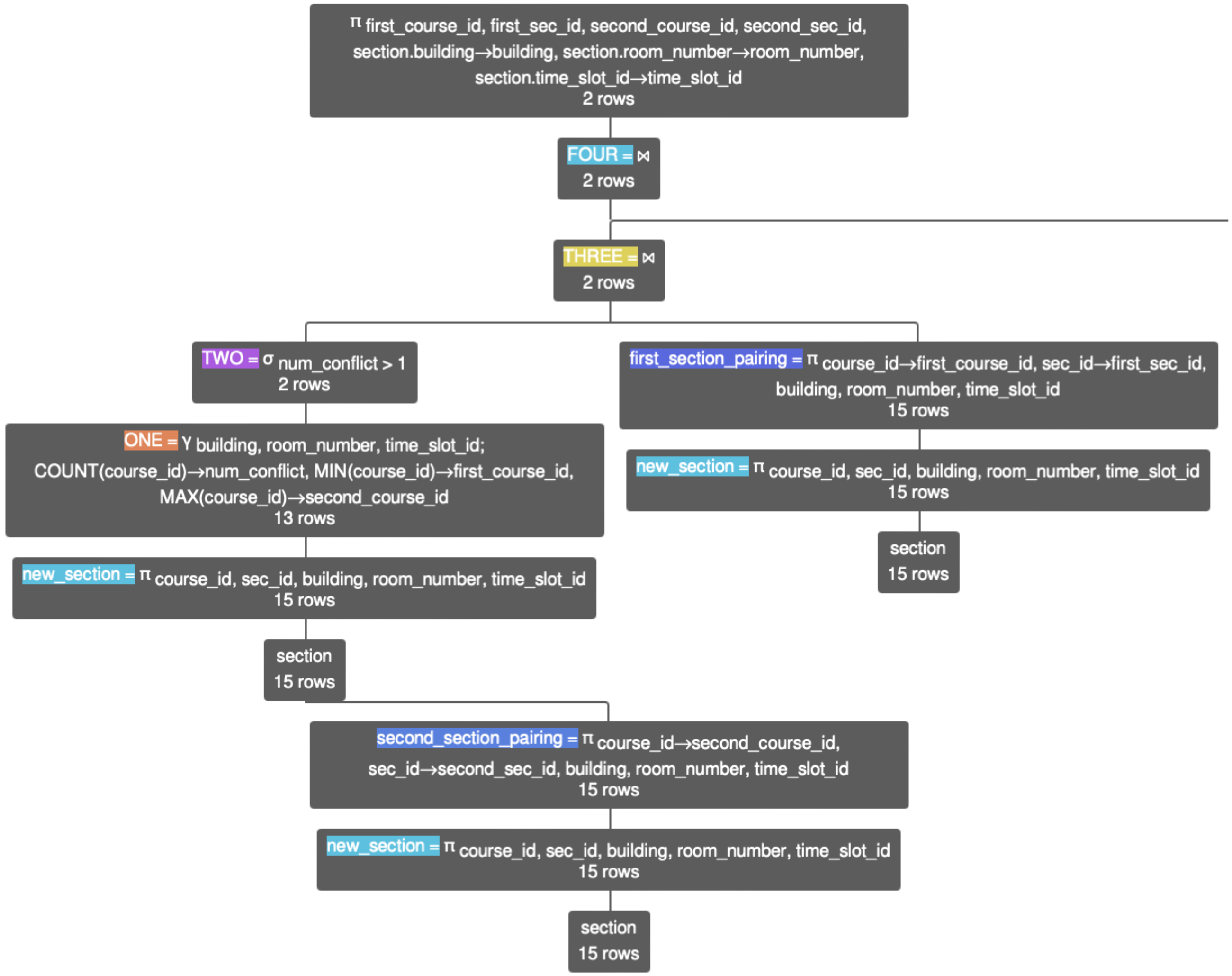
Note: In order to preserve the correct output even for different data:

1. I assume that at most two sections conflict with one another. There is therefore never a need for `third_course_id` etc.
2. I assume that the correct ordering is for `first_course_id` to be chosen in lexicographically ascending order from the conflicting section_ids. Namely, `first_course_id` is earlier in the alphabet/numerical ordering than `second_course_id`

Execution:

In [13]:
```
Image("R3_updated.png")
```

Out[13]:



π first_course_id, first_sec_id, second_course_id, second_sec_id, section.building→building, section.room_number→room_number, section.time_slot_id→time_slot_id ((σ num_conflict > 1 γ building, room_number, time_slot_id; COUNT(course_id)→num_conflict, MIN(course_id)→first_course_id, MAX(course_id)→second_course_id π course_id, sec_id, building, room_number, time_slot_id (section) ⋈ π course_id→first_course_id, sec_id→first_sec_id, building, room_number, time_slot_id π course_id, sec_id, building, room_number, time_slot_id (section)) ⋈ π course_id→second_course_id, sec_id→second_sec_id, building, room_number, time_slot_id π course_id, sec_id, building, room_number, time_slot_id (section))

Execution time: 5 ms

first_course_id	first_sec_id	second_course_id	second_sec_id	building	room_number	time_slot_id
'CS-190'	2	'CS-347'	1	'Taylor'	3128	'A'

'CS-319'	2	'EE-181'	1	'Taylor'	3128	'C'

R3 Execution Result

ER Modeling

Definition to Model

- You're in charge of creating a model for a new music app, Dotify.
- The model has the following entities:
  1. `Artist` has the properties:
     - artist_id (primary key)
     - name
     - description
     - date_joined
  2. `Album` has the properties:
     - album_id (primary key)
     - name
     - release_date
  3. `Song` has the properties:
     - song_id (primary key)
     - title
     - song_length
     - number_of_plays
  4. `User` has the properties:
     - user_id (primary key)
     - name

- bio
- date_joined

5. `Review` has the properties:
    - review_id (primary key)
    - number_of_stars
    - review_text
6. `Playlist` has the properties:
    - playlist_id (primary key)
    - name
    - description

- The model has the following relationships:

1. `Artist-Album` : An artist can have any number of albums. An album belongs to one artist.
2. `Album-Song` : An album can have at least one song. A song is on exactly one album.
3. `Artist-Song` : An artist can have any number of songs. A song has at least one artist.
4. `Album-Review` : An album can have any number of reviews. A review is associated with exactly one album.
5. `User-Review` : A user can write any number of reviews. A review is associated with exactly one user.
6. `User-Playlist` : A user can have any number of playlists. A playlist belongs to exactly one user.
7. `Song-Playlist` : A song can be on any number of playlists. A playlist contains at least one song.

- Other requirements:

1. You may **only** use the four Crow's Foot notations shown in class.
2. A user can leave at most one review per album (you don't need to represent this in your diagram). However, reviews can change over time. Your model must support the ability to keep track of a user's current and previous reviews for an album as well as the dates for the reviews.
3. Playlists can change over time. Your model must support the ability to keep track of current songs in a playlist as well as which songs were on a playlist for what date ranges.
    - You don't need to keep track of a history of when a song was on a playlist (e.g., added Jan 1, then removed Jan 2, then re-added Jan 3, then re-removed Jan 4). You can just track the most recent date range (e.g., added Jan 3, then removed Jan 4).
4. You may not directly link many-to-many relationships. You must use an associative entity.
5. You may (and should) add attributes to the entities and create new entities to fulfill the requirements. **Do not forget**

about foreign keys.

6. You may add notes to explain any reasonable assumptions you make, either on the Lucidchart or below.
   - It would be beneficial, for instance, to document how you implemented requirements 2 and 3.

Assumptions and Documentation

1. I assume that an artist can have zero, one or many albums. Namely, it is possible for an artist to not have put out any albums yet.

2. I have implemented the `Artist-Song` relationship using an associative entity `Recording_Artists` .

3. I have implemented the `Song-Playlist` relationship using an associative entity `Playlist_Library` .

4. I have used the associative entity `Playlist_Library` to implement requirement (3). I have done this by setting the primary key of `Playlist_Library` to be (song_id, playlist_id, date_added). When a song is currently on the playlist, then the date_removed attribute is Null. Otherwise, the date_removed is set to the date the song was taken off the playlist.

   - All current songs on a playlist (with playlist_id X) can be found by querying `Playlist_Library` for all songs where the date_removed attribute is Null
     (e.g. were the model implemented in SQL we could use `select song_id from Playlist_Library WHERE ISNULL(date_removed)=TRUE and playlist_id=X`).

   - Which songs were on a playlist for what date ranges can be found by querying `Playlist_Library` using the desired playlist_id (e.g. X), and setting the desired end points.
     (For example, were the model implemented in SQL, letting the desired range be from `start_date` to `end_date` we could use `select song_id from Playlist_Library WHERE playlist_id=X and date_added ≤ start_date and data_removed ≥ end_date`).

   - The `Playlist_Library` implementation allows us to track the history of when a song was on a playlist and not just the most recent date range. (Since the date_added attribute is part of the Primary key, each time the song is re-added there will be a new row).

   - I assume an implementation of date_added (and date_removed) attributes would use a type similar to time stamp that allows us to differentiate between the times a song is added (e.g. if a song were added deleted and readded on the same day, they would have a different date_added value).

5. I have implemented the `User-Review` relationship using an associative entity `Review History` (the relationship is therefore `Review-Review History` and `Review History-User`). The foreign key constraint that every Review must have a User is therefore implemented in `Review_History` instead of `Review` . A `Review` must have at least 1 corresponding value in `Review History` . A `User` can have zero,1 or many `Review_History` (in general, and for representing revisions/changes to a particular `Review`).

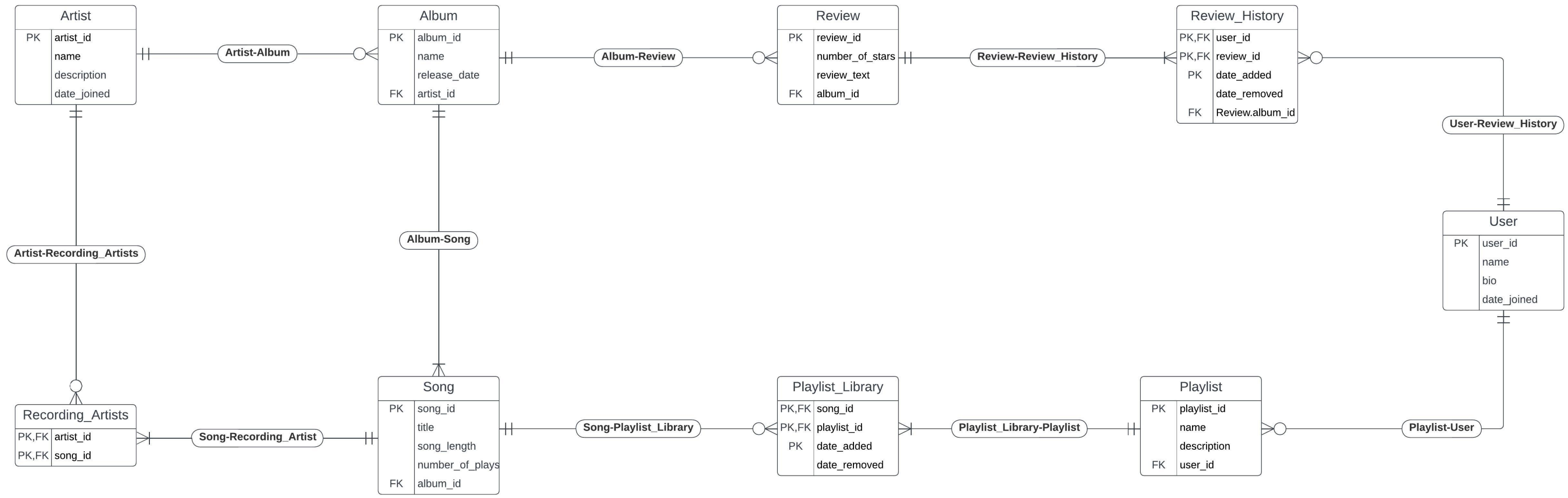
   - I assume every version/update of a review of an album has its own unique review_id.
   - I have added a foreign key to `Review History` of attribute `Review.album_id` referencing the `album_id` attribute in the entity **`Review`** (which in turn references `album_id` in `Album`). I have added this to allow the model to track a user's review history for a paricular album (see ahead). **Note**: The connected foreign keys for album_id across the relationships `Review-Review History` and `Album-Review` ensures that `Review.album_id` will be valid album_id values.
6. I have implemented requirement (2) using an associative entity `Review History` . I have done this by setting the primary key of `Review_History` to be (review_id, user_id, date_added). When a review is the most recent, then the date_removed attribute is Null. Otherwise, the date_removed is set to the date the review was replaced/removed. Similar to bullet 4 above we have:

   - All current reviews for a playlist can be found by querying `Review History` for rows where the date_removed attribute is Null.
   - A User's review history of a particular album containing the current and previous reviews with the dates for the reviews can be found by querying `Review History` for rows with the user_id and Review.album_id attributes set to the desired values.
     (e.g if the desired user has user_id=U and the desired album has album_id=A, were the model implemented in SQL we could use `select * from Review History WHERE user_id=U and Review.album_id=A`).

Diagram:

In [15]:
```
Image("MidtermER2updated.jpeg")
```

Out[15]:



Definition to Model ER Diagram

Model to DDL

- This question tests your ability to convert an ER diagram to DDL.
- Given the ER diagram below (**not your Dotify diagram**), write `create table` statements to implement the model.
  - You should choose appropriate data types, nullness, etc.
  - **You are required to implement the assumptions shown in the diagram.** You can document your other assumptions.
    - The required assumptions can be implemented through correct choices of data types and nullability. You aren't required to write checks or triggers for them.
  - You don't need to execute your statements. You also don't need to worry about details like creating/using a database.

Model to DDL ER Diagram

Answer:

Assumptions and Documentation

- In general, to be safe for all id numbers I have selected the type as VARCHAR(64). As a GUID is 32 characters, I assume the setting of 64 should be more than sufficient to ensure uniqueness where needed.
- As in lecture, I assume that VARCHAR(128) is sufficient for all names (country, first_name, last_name etc..).
- I have set seat_no in Passenger Flight as VARCHAR(5) to allow for 4 digits for the row designation, and an additional character for the seat within the row.
- I have set the foreign key from Passenger Flight to Flight as (airline_id,flight_no), referencing the Primary key of Flight. We want to enforce the constraint that both the airline_id and flight_no are for the same Flight and therefore need the Foreign key to be composite.
- The same applies for the foreign key from Crew Member Flight to Flight.
- I have tried to set non-specified attributes as null or not null by trying to think about what would be a reasonable for a real flight industry setup.
- (I have used the schema Midterm_DDL for testing. The `drop schema..` , `create schema..` replaces the need to check if the individual tables already exist))

```
drop schema if exists Midterm_DDL;
create schema Midterm_DDL;
use Midterm_DDL;

create table Airport
(
    airport_id      VARCHAR(64)  not null,
    airport_iata    CHAR(3)      not null,
    airport_icao    CHAR(4)      not null,
    airport_name    VARCHAR(128) not null,
    airport_country VARCHAR(128) not null,
    constraint Airport_pk
        primary key (airport_id)
);

create table Airline
(
    airline_id      VARCHAR(64)  not null,
```

```
    airline_iata    CHAR(2)      not null,
    airline_icao    CHAR(3)      not null,
    airline_name    VARCHAR(128)  not null,
    active          BOOLEAN      null,
    airline_country VARCHAR(128) not null,
    constraint Airline_pk
        primary key (airline_id)
);

create table Airplane
(
    airplane_tail_no           VARCHAR(64)  not null,
    airplane_manufacturer      VARCHAR(128) null,
    airplane_model             VARCHAR(16)  null,
    airplane_country_of_origin VARCHAR(128) null,
    capacity                   int          null,
    airplane_owner_airline_id  VARCHAR(64)  not null,
    constraint Airplane_pk
        primary key (airplane_tail_no),
    constraint Airplane_owner_airline_id_FK
        foreign key (airplane_owner_airline_id) references Airline (airline_id)
);

create table `Crew Member`
(
    pilot_id            VARCHAR(64)  not null,
    employer_airline_id VARCHAR(64)  not null,
    first_name          VARCHAR(128) not null,
    last_name           VARCHAR(128) not null,
    birthdate           DATE         not null,
    country_of_origin   VARCHAR(128) not null,
    passport_no         VARCHAR(64)  not null,
    start_date          DATE         not null,
    end_date            DATE         null,
    constraint `Crew Member_pk`
        primary key (pilot_id),
    constraint employer_airline_id___fk
        foreign key (employer_airline_id) references Airline (airline_id)
);
```

```
create table Passenger
(
    passenger_id      VARCHAR(64)  not null,
    first_name        VARCHAR(128) not null,
    last_name         VARCHAR(128) not null,
    birthdate         Date         not null,
    country_of_origin VARCHAR(128) not null,
    passport_no       VARCHAR(64)  not null,
    constraint Passenger_pk
        primary key (passenger_id)
);

create table Flight
(
    airline_id         VARCHAR(64) not null,
    flight_no          VARCHAR(64) not null,
    departure_airport  VARCHAR(64) not null,
    arrival_airport    VARCHAR(64) not null,
    departure_datetime DATETIME    null,
    arrival_datetime   DATETIME    null,
    airplane_id        VARCHAR(64) not null,
    constraint Flight_pk
        primary key (airline_id, flight_no),
    constraint airline_id___fk
        foreign key (airline_id) references Airline (airline_id),
    constraint airplane_id___fk
        foreign key (airplane_id) references Airplane (airplane_tail_no),
    constraint arrival_airport___fk
        foreign key (arrival_airport) references Airport (airport_id),
    constraint departure_airport___fk
        foreign key (departure_airport) references Airport (airport_id)
);

create table `Crew Member Flight`
(
    pilot_id      varchar(64)                                                   not
null,
    airline_id    varchar(64)                                                   not
```

```
null,
    flight_no     varchar(64)                                                     not
null,
    crew_position enum ('pilot', 'copilot', 'flight engineer', 'flight navigator') null,
    primary key (pilot_id, airline_id, flight_no),
    constraint `Crew Member Flight_Crew Member_pilot_id_fk`
        foreign key (pilot_id) references Crew Member (pilot_id),
    constraint `Crew Member Flight_Flight_airline_id_flight_no_fk`
        foreign key (airline_id, flight_no) references Flight (airline_id, flight_no)
);

create table `Passenger Flight`
(
    passenger_id VARCHAR(64)                              not null,
    airline_id   VARCHAR(64)                              not null,
    flight_no    VARCHAR(64)                              not null,
    travel_class ENUM ('First', 'Business', 'Economy') null,
    seat_no      VARCHAR(4)                               null,
    on_flight    BOOLEAN                                  not null,
    constraint `Passenger Flight_pk`
        primary key (passenger_id, airline_id, flight_no),
    constraint `Passenger Flight_Flight_airline_id_flight_no_fk`
        foreign key (airline_id, flight_no) references Flight (airline_id, flight_no),
    constraint `Passenger Flight_Passenger_passenger_id_fk`
        foreign key (passenger_id) references Passenger (passenger_id)
);
```

Data and Schema Cleanup

Setup

- There are several issues with the `classicmodels` schema. Two issues are:

    - Having programs or users enter country names for `customers.country` is prone to error.
    - `products.productCode` is clearly not an atomic value.
- The following code does the following:

    1. Creates a schema for this question
    2. Creates copies of `classicmodels.customers` and `classicmodels.products`
    3. Loads a table of ISO country codes

In [11]:
```
%%sql

drop schema if exists classicmodels_midterm;
create schema classicmodels_midterm;
use classicmodels_midterm;

create table customers as select * from classicmodels.customers;
create table products as select * from classicmodels.products;
```

```
 * mysql+pymysql://root:***@localhost
4 rows affected.
1 rows affected.
0 rows affected.
122 rows affected.
110 rows affected.
```

Out[11]: []

In [12]:
```
iso_df = pandas.read_csv('./wikipedia-iso-country-codes.csv')
iso_df.to_sql('countries', schema='classicmodels_midterm',
              con=engine, index=False, if_exists="replace")
```

Out[12]: 246

In [13]:
```
%%sql

alter table countries
    change `English short name lower case` short_name varchar(64) null;

alter table countries
    change `Alpha-2 code` alpha_2_code char(2) null;
```

```
alter table countries
    change `Alpha-3 code` alpha_3_code char(3) not null;

alter table countries
    change `Numeric code` numeric_code smallint unsigned null;

alter table countries
    change `ISO 3166-2` iso_text char(13) null;

alter table countries
    add primary key (alpha_3_code);

select * from countries limit 10;
```

```
 * mysql+pymysql://root:***@localhost
246 rows affected.
246 rows affected.
246 rows affected.
246 rows affected.
246 rows affected.
0 rows affected.
10 rows affected.
```

Out[13]:

short_name	alpha_2_code	alpha_3_code	numeric_code	iso_text
Aruba	AW	ABW	533	ISO 3166-2:AW
Afghanistan	AF	AFG	4	ISO 3166-2:AF
Angola	AO	AGO	24	ISO 3166-2:AO
Anguilla	AI	AIA	660	ISO 3166-2:AI
Åland Islands	AX	ALA	248	ISO 3166-2:AX
Albania	AL	ALB	8	ISO 3166-2:AL
Andorra	AD	AND	20	ISO 3166-2:AD
Netherlands Antilles	AN	ANT	530	ISO 3166-2:AN
United Arab Emirates	AE	ARE	784	ISO 3166-2:AE
Argentina	AR	ARG	32	ISO 3166-2:AR

DE1

- There are four values in `customers.country` that do not appear in `countries.short_name`.
- Write a query that finds these four countries.
  - Hint: Norway should be one of these countries.

In [14]:
```
%%sql
select country
from customers
WHERE customers.country NOT IN (select short_name from countries)
GROUP BY country;
```

```
 * mysql+pymysql://root:***@localhost
4 rows affected.
```

Out[14]:

country
USA
Norway
UK
Russia

DE2

- `Norway` actually does appear in `countries.short_name`. The reason it appeared in DE1 is because there are two spaces after the name (`Norway__` instead of `Norway`).
- The mapping for the other countries is:

customers.country	countries.short_name
USA	United States
UK	United Kingdom
Russia	Russian Federation

- Write `update table` statements to correct the values in `customers.country` so that all the values in that attribute appear in `countries.short_name`.

In [15]:
```
%%sql
UPDATE customers
SET country = CASE
                  WHEN country = 'Norway  ' then 'Norway'
                  WHEN country = 'USA' then 'United States'
                  WHEN country = 'UK' then 'United Kingdom'
                  WHEN country = 'Russia' then 'Russian Federation'
                  Else country
    END;
```

```
 * mysql+pymysql://root:***@localhost
122 rows affected.
```

Out[15]: []

DE3

- The PK of `countries` is `alpha_3_code`. We want that as a FK in `customers`.

1. Create a column `customers.iso_code`
2. Set `customers.iso_code` as a FK that references `countries.alpha_3_code`
3. Fill `customers.iso_code` with the appropriate data based on `customers.country`
4. Drop `customers.country`
5. Create a view `customers_country` of form `(customerNumber, customerName, country, iso_code)`

Bonus point: I would ask you to create an index on `customers.iso_code`, but this is actually already done for us. When was an index created on `customers.iso_code`?

Answer An index was created on `customers.iso_code` in step (2) when we created the FK. MySQL automatically created the index to more efficiently implement the foreign key constraint.

In [16]:
```
%%sql
alter table customers
    add iso_code CHAR(3) not null;
```

```
UPDATE customers, countries
SET customers.iso_code = countries.alpha_3_code
WHERE customers.country = countries.short_name;

alter table customers
    add constraint customers_customers__fk
        foreign key (iso_code) references countries (alpha_3_code);

alter table customers
    drop column country;

create definer = root@localhost view customers_country as
select customerNumber, customerName, short_name as country, iso_code
FROM customers
         JOIN classicmodels_midterm.countries c on c.alpha_3_code = customers.iso_code;
```

```
 * mysql+pymysql://root:***@localhost
0 rows affected.
122 rows affected.
122 rows affected.
0 rows affected.
0 rows affected.
```

Out[16]: []

DE4

- To test your code, output a table that shows the number of customers from each country.
- You should use your `customers_country` view.
- Your table should have the following attributes:
    - `country_iso`
    - `number_of_customers`
- Order your table from greatest to least `number_of_customers` .
- Show only the first 10 rows.

In [17]:
```
%%sql
SELECT iso_code as country_iso, COUNT(DISTINCT customerNumber) as number_of_customers
FROM customers_country
```

```
GROUP BY iso_code
ORDER BY number_of_customers DESC
LIMIT 10;
```

```
* mysql+pymysql://root:***@localhost
10 rows affected.
```

Out[17]:

country_iso	number_of_customers
USA	36
DEU	13
FRA	12
ESP	7
GBR	5
AUS	5
ITA	4
NZL	4
FIN	3
CAN	3

DE5

- `products.productCode` appears to be 3 separate values joined by an underscore.
  - I have no idea what the values mean, but let's pretend we do know for the sake of this question.
- Write `alter table` statements to create 3 new columns: `product_code_letter` , `product_code_scale` , and `product_code_number` .
  - Choose appropriate data types. `product_code_letter` should always be a single letter.

In [18]:

```
%%sql
alter table products
    add product_code_letter CHAR not null,
    add product_code_scale VARCHAR(3) not null,
```

```
    add product_code_number VARCHAR(4) not null;
```

```
 * mysql+pymysql://root:***@localhost
0 rows affected.
```

Out[18]: []

DE6

- As an example, for the product code `S18_3856`, the product code letter is `S`, the product code scale is `18`, and the product code number is `3856`.
  - I know the product code scale doesn't always match `products.productScale`. Let's ignore this for now.

1. Populate `product_code_letter`, `product_code_scale`, and `product_code_number` with the appropriate values based on `productCode`.
2. Set the PK of `products` to `(product_code_letter, product_code_scale, product_code_number)`.
3. Drop `productCode`.

In [19]:

```
%%sql
UPDATE products
SET product_code_letter = substr(productCode, 1, 1),
    product_code_scale  = substr(substring_index(products.productCode, '_', 1), 2,
                                 length(substring_index(products.productCode, '_', 1))),
    product_code_number = substring_index(products.productCode, '_', -1);

alter table products
    add constraint products_pk
        primary key (product_code_letter, product_code_scale, product_code_number);

alter table products
    drop column productCode;
```

```
 * mysql+pymysql://root:***@localhost
110 rows affected.
0 rows affected.
0 rows affected.
```

Out[19]: []

DE7

- To test your code, output a table that shows the products whose `product_code_scale` doesn't match `productScale` .
- Your table should have the following attributes:
  - `product_code_letter`
  - `product_code_scale`
  - `product_code_number`
  - `productScale`
  - `productName`
- Order your table on `productName` .

In [20]:
```
%%sql
SELECT product_code_letter, product_code_scale, product_code_number, productScale, productName
FROM products
WHERE product_code_scale != substring_index(products.productScale, ':', -1)
ORDER BY productName;
```

```
 * mysql+pymysql://root:***@localhost
6 rows affected.
```

Out[20]:

product_code_letter	product_code_scale	product_code_number	productScale	productName
S	24	3856	1:18	1956 Porsche 356A Coupe
S	24	4620	1:18	1961 Chevrolet Impala
S	12	3148	1:18	1969 Corvair Monza
S	700	2824	1:18	1982 Camaro Z28
S	700	3167	1:72	F/A 18 Hornet 1/72
S	18	2581	1:72	P-51-D Mustang

SQL

- Use the `classicmodels` database for these questions.
- The suggestions on which tables to use are hints, not requirements.
- All your answers should be a single select statement. **You may not create a new table.**
  - Subqueries (selects within a select) and the `with` keyword are fine. Just don't use the `create` keyword.

In [22]:
```
%sql use classicmodels
```

```
 * mysql+pymysql://root:***@localhost
0 rows affected.
```

Out[22]: []

SQL1

- Write a query that produces a table of form `(productName, productLine, productVendor, totalRevenue)`.
  - Attribute names should match exactly.
  - The `totalRevenue` for a product is the sum of `quantityOrdered*priceEach` across all the rows the product appears in in `orderdetails`.
  - You should consider all orders, regardless of `orders.status`.
- Only include products with `totalRevenue` greater than $150,000.
- Order your output on `totalRevenue` descending.
- You should use the `products` and `orderdetails` tables.

In [17]:
```
%%sql
select productName,
       productLine,
       productVendor,
       SUM(orderdetails.quantityOrdered * orderdetails.priceEach) as totalRevenue
From products
         join orderdetails using (productCode)
GROUP BY productCode, productName, productLine, productVendor
HAVING totalRevenue > 150000
ORDER BY totalRevenue DESC;
```

```
 * mysql+pymysql://root:***@localhost
6 rows affected.
```

Out[17]:

productName	productLine	productVendor	totalRevenue
1992 Ferrari 360 Spider red	Classic Cars	Unimax Art Galleries	276839.98
2001 Ferrari Enzo	Classic Cars	Second Gear Diecast	190755.86
1952 Alpine Renault 1300	Classic Cars	Classic Metal Creations	190017.96
2003 Harley-Davidson Eagle Drag Bike	Motorcycles	Red Start Diecast	170686.00
1968 Ford Mustang	Classic Cars	Autoart Studio Design	161531.48
1969 Ford Falcon	Classic Cars	Second Gear Diecast	152543.02

SQL2

- Write a query that produces a table of form `(productCode, productName, productVendor, customerCount)` .
  - Attribute names should match exactly.
  - `customerCount` is the number of **distinct** customers that have bought the product.
    - Note that the same customer may buy a product multiple times. This only counts as one customer in the product's `customerCount` .
  - You should consider all orders, regardless of `status` .
- Order your table from largest to smallest `customerCount` , then on `productCode` alphabetically.
- Only show the first 10 rows.
- You should use the `orders` and `orderdetails` tables.

In [29]:

```
%%sql
select productCode,
       productName,
       productVendor,
       Count(Distinct customerNumber) as customerCount
From (orders join orderdetails using (orderNumber))
         join products using (productCode)
```

```
GROUP BY productCode
ORDER BY customerCount DESC, productCode ASC
LIMIT 10;
```

```
* mysql+pymysql://root:***@localhost
10 rows affected.
```

Out[29]:

productCode	productName	productVendor	customerCount
S18_3232	1992 Ferrari 360 Spider red	Unimax Art Galleries	40
S10_1949	1952 Alpine Renault 1300	Classic Metal Creations	27
S10_4757	1972 Alfa Romeo GTA	Motor City Art Classics	27
S18_2957	1934 Ford V8 Coupe	Min Lin Diecast	27
S72_1253	Boeing X-32A JSF	Motor City Art Classics	27
S10_1678	1969 Harley Davidson Ultimate Chopper	Min Lin Diecast	26
S10_2016	1996 Moto Guzzi 1100i	Highway 66 Mini Classics	26
S18_1662	1980s Black Hawk Helicopter	Red Start Diecast	26
S18_1984	1995 Honda Civic	Min Lin Diecast	26
S18_2949	1913 Ford Model T Speedster	Carousel DieCast Legends	26

SQL3

- Write a query that produces a table of form `(customerName, month, year, monthlyExpenditure, creditLimit)`.
  - Attribute names should match exactly.
  - `monthlyExpenditure` is the total amount of payments made by a customer in a specific month and year based on the `payments` table.
    - Some customers have never made any payments ever. For these customers, `monthlyExpenditure` should be 0. `month` and `year` can be null.
- Only show rows where `monthlyExpenditure` exceeds `creditLimit` **or** the customer has never made any payments ever (so `month` and `year` should be null for these rows).

- Order your table on `monthlyExpenditure` descending, then on `customerName` alphabetically.
- Only show the first 10 rows.
- You should use the `payments` and `customers` tables.

In [60]:

```
%%sql
select customerName,
       Month(paymentDate) as month,
       YEAR(paymentDate) as year,
       IFNULL(SUM(amount),0) as monthlyExpenditure,
       creditLimit
FROM customers LEFT OUTER JOIN payments on customers.customerNumber = payments.customerNumber
GROUP BY customers.customerNumber,month,year,creditLimit
HAVING creditLimit < monthlyExpenditure or ISNULL(month)
ORDER BY monthlyExpenditure DESC, customerName ASC
LIMIT 10;
```

```
 * mysql+pymysql://root:***@localhost
10 rows affected.
```

Out[60]:

customerName	month	year	monthlyExpenditure	creditLimit
Dragon Souveniers, Ltd.	12	2003	105743.00	103800.00
American Souvenirs Inc	None	None	0.00	0.00
ANG Resellers	None	None	0.00	0.00
Anton Designs, Ltd.	None	None	0.00	0.00
Asian Shopping Network, Co	None	None	0.00	0.00
Asian Treasures, Inc.	None	None	0.00	0.00
BG&E Collectables	None	None	0.00	0.00
Cramer Spezialitäten, Ltd	None	None	0.00	0.00
Der Hund Imports	None	None	0.00	0.00
Feuer Online Stores, Inc	None	None	0.00	0.00

SQL4

- Write a query that produces a table of form `(productCode, productName, productLine, productVendor, productDescription)`.
  - Attribute names should match exactly.
- **You should only keep products that have never been ordered by a French customer.**
  - You should consider all orders, regardless of `status`.
- Order your table on `productCode`.
- You should use the `customers`, `orders`, and `orderdetails` tables.

In [23]:
```
%%sql
WITH French_customers as (select customerNumber
                          from customers
                          where country = 'France'),
     French_orders as (select orderNumber
                       from French_customers
                                join orders using (customerNumber)),
     French_products as (select productCode
                         from French_orders
                                  join orderdetails using (orderNumber)
                         GROUP BY productCode),
     NON_French as (select productCode from products where productCode NOT IN (select productCode from Frer
select productCode, productName, productLine, productVendor, productDescription
from NON_French
         join products using (productCode)
ORDER BY productCode;
```

```
 * mysql+pymysql://root:***@localhost
2 rows affected.
```

Out[23]:

productCode	productName	productLine	productVendor	productDescription
S18_3233	1985 Toyota Supra	Classic Cars	Highway 66 Mini Classics	This model features soft rubber tires, working steering, rubber mud guards, authentic Ford logos, detailed undercarriage, opening doors and hood, removable split rear gate, full size spare mounted in bed, detailed interior with opening glove box
S18_4027	1970 Triumph Spitfire	Classic Cars	Min Lin Diecast	Features include opening and closing doors. Color: White.

SQL5

- A customer can have a sales rep employee.
- Corporate is deciding which employees to give raises to.
  - A raise is given for the reason `customers` if an employee has 8 or more customers.
  - A raise is given for the reason `orders` if the total number of orders made by customers associated with an employee is 30 or greater.
    - You should consider all orders, regardless of `status` .
  - A raise is given for the reason `both` if both conditions above are true.
- Write a query that produces a table of form `(firstName, lastName, totalCustomers, totalCustomerOrders, raiseBecause)` .
  - Attribute names should match exactly.
  - `firstName` and `lastName` are for the employee.
  - `totalCustomers` is the total number of customers associated with an employee.
  - `totalCustomerOrders` is the total number of orders made by customers associated with an employee.
  - `raiseBecause` is one of `customers` , `orders` , and `both` .
- Your table should only show employees eligible for raises, i.e., `raiseBecause` should not be null.
- Order your table on `firstName` .
- You should use the `customers` , `orders` , and `employees` tables.

NOTE: I assumed we should only give a raise for the reason `customers` if an employee has 8 or more customers **who actually placed an order**.

For the given data this does not make a difference either way: there are 2 customers (#168 and #376) with associated employees (#1286 and #1702 respectively) who made no orders, but neither employee #1286 nor #1702 would earn a raise even if these customers were to be included.

Note: For clarity, I included "DISTINCT" in `Count(DISTINCT orderNumber)` in order_per_customer. This is not strictly necessary as orderNumber is a primary key of the table orders.

In [23]:

```
%%sql
WITH order_per_customer(customerNumber, num_orders_per_customer, salesRepEmployeeNumber)
         as (select customerNumber, Count(DISTINCT orderNumber), salesRepEmployeeNumber
             from orders
                      join customers using (customerNumber)
             GROUP BY customerNumber),
     employees_customers as (select *
                             from order_per_customer
                                      join employees on salesRepEmployeeNumber = employeeNumber),
     computed (firstName, lastName, totalCustomers, totalCustomerOrders)
        as (select firstName,
            lastName,
            COUNT(DISTINCT customerNumber),
            SUM(num_orders_per_customer)
            from employees_customers
            GROUP BY employeeNumber, firstName, lastName),

     raises(firstName, lastName, totalCustomers, totalCustomerOrders, raiseBecause)
            as (select firstName,
                        lastName,
                        totalCustomers,
                        totalCustomerOrders,
                        CASE
                            WHEN totalCustomerOrders >= 30 and totalCustomers >= 8
                                then 'both'
                            WHEN totalCustomerOrders >= 30
                                then 'orders'
                            WHEN totalCustomers >= 8
                                then 'customers'
                            END
```

```
                             from computed)
select *
from raises
WHERE ISNULL(raiseBecause) = FALSE
ORDER BY firstName;
```

```
* mysql+pymysql://root:***@localhost
6 rows affected.
```

Out[23]:

firstName	lastName	totalCustomers	totalCustomerOrders	raiseBecause
Barry	Jones	9	25	customers
George	Vanauf	8	22	customers
Gerard	Hernandez	7	43	orders
Larry	Bott	8	22	customers
Leslie	Jennings	6	34	orders
Pamela	Castillo	10	31	both